15-110 Quiz 4 Review Session

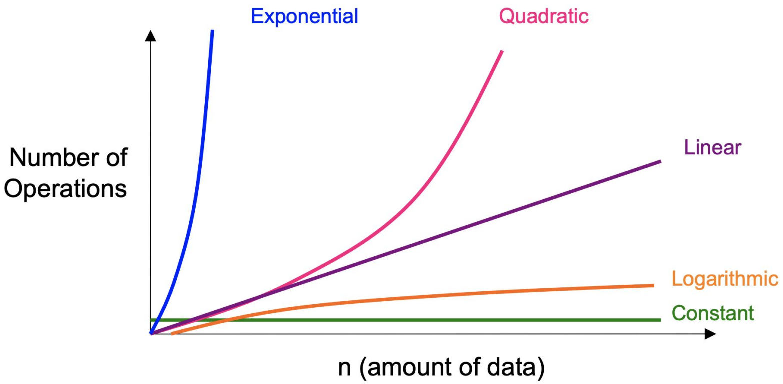
Led by: Stephen and Otto

Gameplan

- Review the major topics
  - Big O
  - Trees
  - Graphs
  - Tractability
- Go over some practice problems
- Q + A

Big O

- A way for programmers to group certain functions into different families based on their speed/efficiency
- Measure based on worst case
- Ignore lower order terms!



f(n)	Big-O
n	O(n)
32n + 23	O(n)
$5n^2 + 6n - 8$	$O(n^2)$
18 log(n)	O(log n)

Common Themes in Big O

- For loops
  - Check how many times the for loop is running
    - If it's related to the input (i.e. len(lst)), then it's O(n)
    - If it's looping a constant amount (for i in range(20)), it's O(1)
- While loops
  - Check how the while loop is increasing/decreasing the iterator variable
    - If it's using addition/subtraction, it's O(n)
    - If it's using multiplication/division, it's O(log n)
- Most built in functions/methods (len, .append(), indexing into an array, etc.) are O(1) **UNLESS OTHERWISE SPECIFIED**
  - .find(), in, and other methods/functions that search through are O(n)

Strategy to Big O

1. Go through a function line by line
2. Find the Big O value of each individual line
   a. Most lines are O(1)!
3. Once you're done, go through to find the total Big O
   a. If it's one the same indentation levels, add up their big O's
      i. O(1) + O(1) + O(1) = O(3)
   b. If a line is indented, or nested, in another line, multiply their big O's
      i. O(n) * O(n) = O(n^2)
4. At the end, make sure to simplify whatever value you get to fit into a function family
   a. O(3n^2 + 2n + 6) → O(n^2)

Big O Example

```python
# s is a string larger than 10 characters
def mur(s):
    result = ""
    for i in range(10):
        result += s[i]

    return result
```

Big O Example

```python
# L is a list of integers
def allPossiblePairs(L):
    result = []
    for x in L:
        for y in L:
            result += [[x, y]]
    return result
```

Big O Example

```python
# n is a positive integer
def foo(n):
    result = 0
    while (n > 0):
        n = n // 2
        result += 1
    return result
```

Tractability

- A problem is said to be tractable if it has a reasonably efficient runtime:
    - O(1), O(log n), O(n log n), O(n^2), O(n^10000)
    - ^ Polynomial time
- Intractable:
    - O(2^n), O(n!), O(k^n)
    - ^ Bigger than polynomial time

Brute Force Algorithms

- Brute force algorithms check every possible solution.
  - Ex. Testing every subset of [1,2,3] for subset sum ====>
  - Other ones: travelling salesperson, puzzle-solving, and exam scheduling
- Generally are intractable solutions.

- []
- [1]

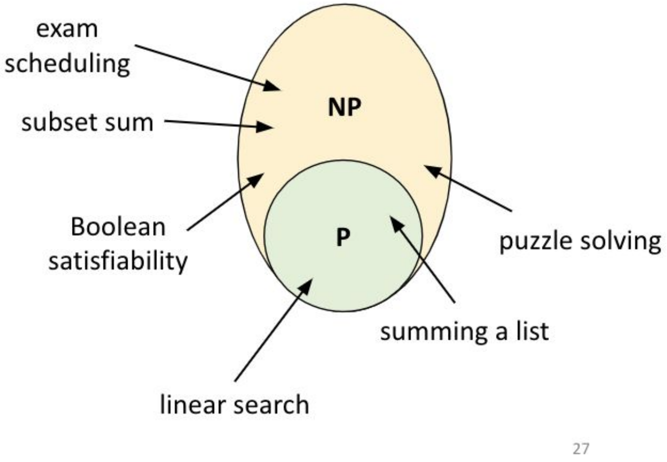
- [2]
- [1, 2]

- [3]
- [1, 3]

- [2, 3]
- [1, 2, 3]

P vs. NP

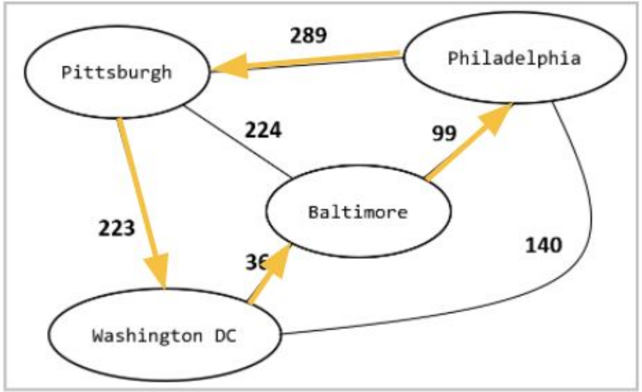
	P	NP
Verifying	Tractable	Tractable
Solving	Tractable	?



Why isn’t traveling salesperson in this diagram?

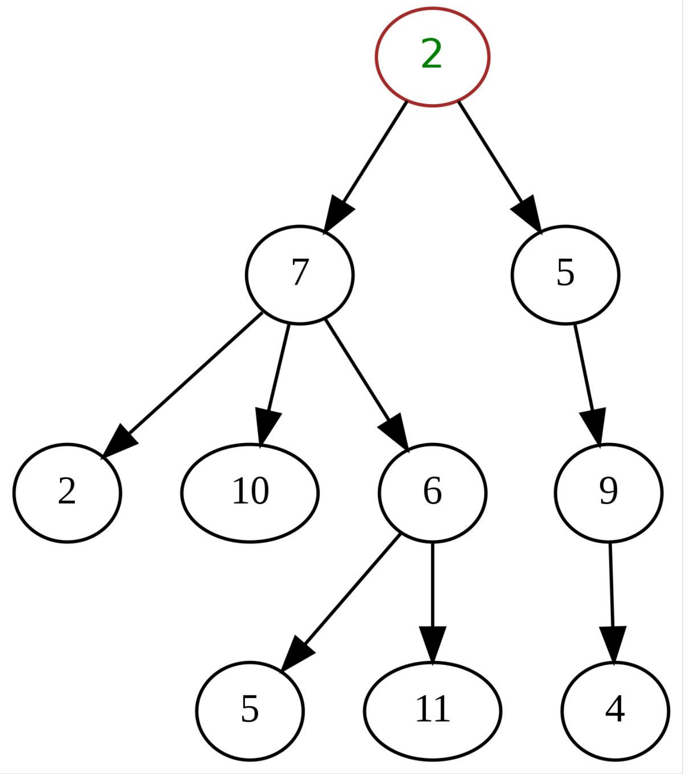
Heuristics

- Shortcuts to find a solution that is not the best, but is close.
  - Finding the best solution often takes a really long time
  - Finding a good solution is often much easier

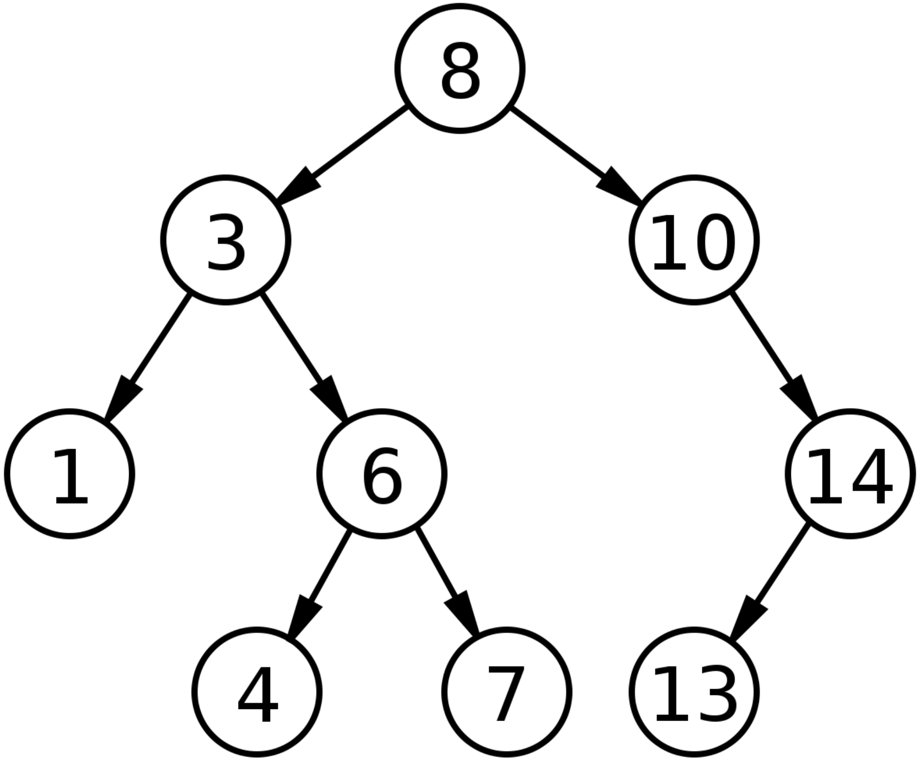


Trees

- Another different data structure
- Hierarchical structure (up, down)
- Uses nodes, and each node has a value
- Nodes connected below a node are the children, and the node above is the parent
- Top node is root, nodes with no children are leaves



Trees Example



Root: 8

Node: all of them! (1, 3, 4, 6…13)

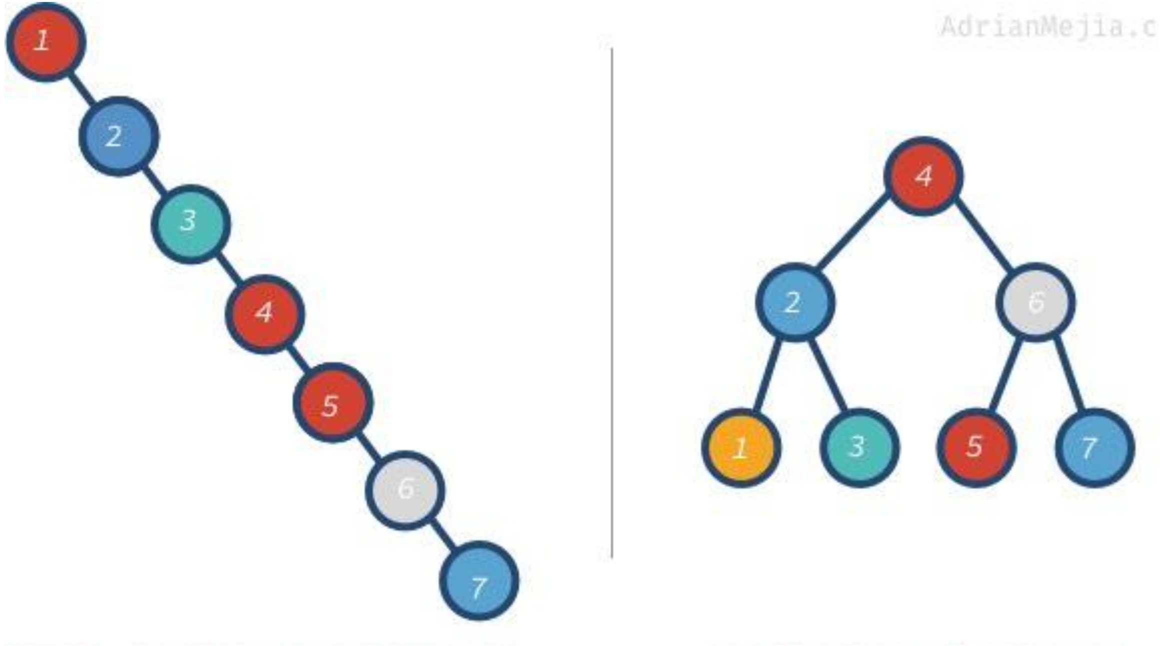
Children (of 8): 3, 10

Leaves: 1, 4, 7, 13

Binary Search Trees

- A specific type of tree which allows for easy search of values (binary search! It's in the name!)
- The main rule for BSTs
    - Everything to the **left** of a node must be **less** than the value at that node, and everything to the **right** of the node must be **greater** than the value at that node

Balanced vs. Unbalanced Search Trees



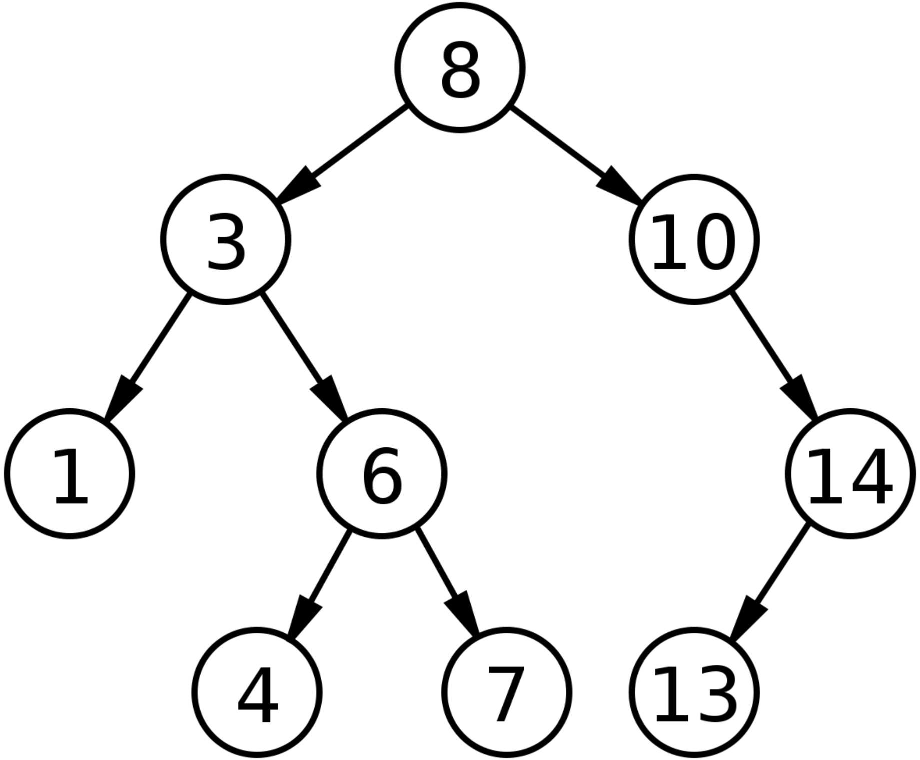
Trees in Code

- Store in a dictionary!
  - Three keys: contents, left, right
    - Contents is the value at that node
    - Left and right are either another dictionary containing the same three keys, or None, meaning there is no child.

```
{ "contents" : nodeValue,
  "left"     : leftChildSubtree,
  "right"    : rightChildSubtree }
```

Trees Example

Say we are given a tree, where each node has two children (not necessarily a BST, but you can think of it like that). Write a recursive function addOdds(tree) that adds all of the odd leaves and returns the sum of all the odd leaves.



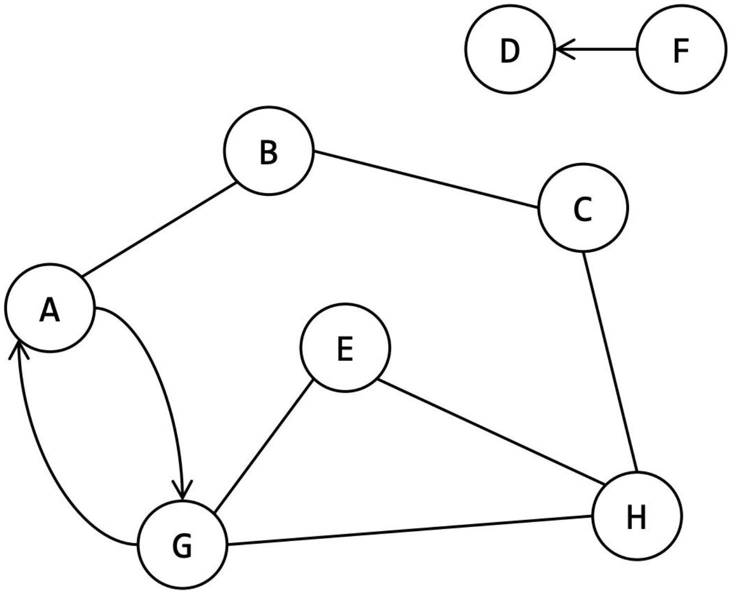
```python
def addOdds(tree):
    if tree["left"] == None and tree["right"] == None:
        # Base case: We are at a leaf node
        if tree["contents"] % 2 == 1:
            return tree["contents"]
        else:
            return 0
    else:
        # Recursive case: We are not at a leaf node
        result = 0
        if tree["left"] != None:
            result += addOdds(tree["left"])

        if tree["right"] != None:
            result += addOdds(tree["right"])

        return result
```

Graphs

- Similar to trees – nodes connected to other nodes
- Less restrictions – any node can be connected to any other node, no longer follows a hierarchical structure
- Graphs have **edges** – edges are the connections between the nodes
    - Sometimes, edges can have **weights**, which is just a number associated with the edge
    - Edges can also be **directed** or **undirected**



Graphs in Code

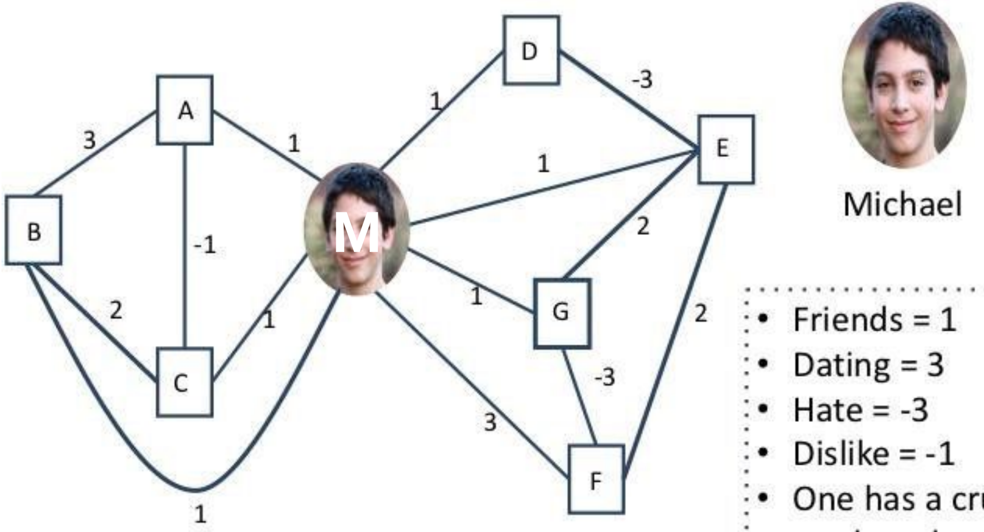
```
{ node : [ [neighborValue, weight] ],
  ... }
```

- Treated as a dictionary!
  - Stored slightly differently if the graph is weighted or not
  - Loop through all nodes with a for each loop (same as keys in dictionary)

```
{ nodeValue : [ neighborValue ],
  ... }
```

Graph Problem

Say we are given a certain person in our graph, in this case, Michael. Write a function findFriendsList(person, g) that takes a person and a graph, and returns all the friends of that person in the form of a list.

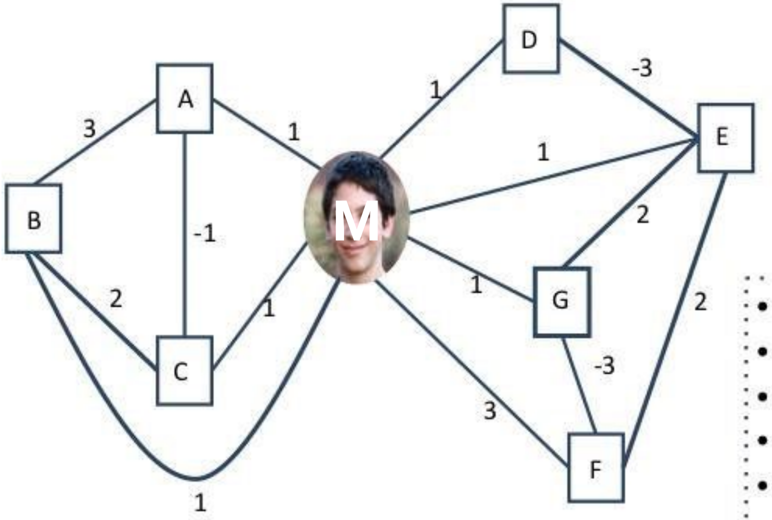


- Friends = 1
- Dating = 3
- Hate = -3
- Dislike = -1
- One has a crush on the other = 2
- No relationship stated = 0

```python
def getFriends(person, g):
    friendList = []
    for relation in g[person]:
        if relation[1] == 1:
            # This is a friend :)
            friendList += [relation[0]]
    return friendList
```

Graph Problem

Write a function getAllCouples(g) that takes in a graph, and returns a list of all the couples in the graph, stored together as a 2d list.



Michael

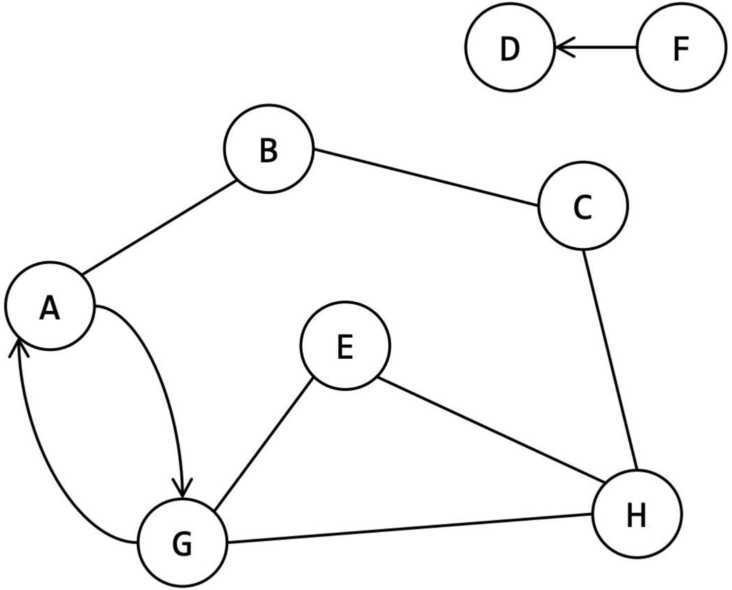
- Friends = 1
- Dating = 3
- Hate = -3
- Dislike = -1
- One has a crush on the other = 2
- No relationship stated = 0

```python
def getAllCouples(g):
    couples = []
    for person in g:
        for relation in g[person]:
            if relation[1] == 3:
                # They are dating
                couple = [person, relation[0]]
                couples += [couple]

    # NOTE: This will contain duplicates of each couple
    return couples
```

Breadth vs. Depth First Search

- Different ways of searching for a given node on a graph
  - BFS: start at a node, go through every one of its neighbors, and then go to the neighbors after, etc. until found/looked through everything
  - DFS: start at a node, keep searching down one path until you can/can't find something, start from start again and repeat



That’s it!

Any questions?